

## 04 | 编译阶段能做什么：属性和静态断言

2020-05-14 罗剑锋

罗剑锋的C++实战笔记

[进入课程 >](#)



讲述：Chrono

时长 13:05 大小 12.00M



你好，我是 Chrono。

前面我讲了 C++ 程序生命周期里的“编码阶段”和“预处理阶段”，它们的工作主要还是“文本编辑”，生成的是**人类可识别的源码**（source code）。而“编译阶段”就不一样了，它的目标是**生成计算机可识别的机器码**（machine instruction code）。

今天，我就带你来看看在这个阶段能做些什么事情。




**编译阶段编程**

编译是预处理之后的阶段，它的输入是（经过预处理的）C++ 源码，输出是**二进制可执行文件**（也可能是汇编文件、动态库或者静态库）。这个处理动作就是由编译器来执行的。

和预处理阶段一样，在这里你也可以“面向编译器编程”，用一些指令或者关键字让编译器按照你的想法去做一些事情。只不过，这时你要面对的是庞大的 C++ 语法，而不是简单的文本替换，难度可以说是高了好几个数量级。

编译阶段的特殊性在于，它看到的都是 C++ 语法实体，比如 typedef、using、template、struct/class 这些关键字定义的类型，而不是运行阶段的变量。所以，这时的编程思维方式与平常大不相同。我们熟悉的是 CPU、内存、Socket，但要去理解编译器的运行机制、知道怎么把源码翻译成机器码，这可能就有点“强人所难”了。

比如说，让编译器递归计算斐波那契数列，这已经算是一个比较容易理解的编译阶段数值计算用法了：

 复制代码

```
1  template<int N>
2  struct fib                                // 递归计算斐波那契数列
3  {
4      static const int value =
5          fib<N - 1>::value + fib<N - 2>::value;
6  };
7
8  template<>
9  struct fib<0>                            // 模板特化计算fib<0>
10 {
11     static const int value = 1;
12 };
13
14 template<>
15 struct fib<1>                            // 模板特化计算fib<1>
16 {
17     static const int value = 1;
18 };
19
20 // 调用后输出2, 3, 5, 8
21 cout << fib<2>::value << endl;
22 cout << fib<3>::value << endl;
23 cout << fib<4>::value << endl;
24 cout << fib<5>::value << endl;
```

对于编译器来说，可以在一瞬间得到结果，但你要搞清楚它的执行过程，就得在大脑里把 C++ 模板特化的过程走一遍。整个过程无法调试，完全要靠自己推导，特别“累人”。（你也可以把编译器想象成是一种特殊的“虚拟机”，在上面跑的是只有编译器才能识别、处理的代码。）

简单的尚且如此，那些复杂的就更不用说了。所以，今天我不去讲那些太过于“烧脑”的知识了，而是介绍两个比较容易理解的编译阶段技巧：属性和静态断言，让你能够立即用得上，效果也是“立竿见影”。

## 属性 (attribute)

“预处理编程”[🔗这一讲](#)提到的 `#include`、`#define` 都是预处理指令，是用来控制预处理器的。那么问题就来了，有没有用来控制编译器的“编译指令”呢？

虽然编译器非常聪明，但因为 C++ 语言实在是太复杂了，偶尔它也会“自作聪明”或者“冒傻气”。如果有这么一个东西，让程序员来手动指示编译器这里该如何做、那里该如何做，就有可能生成更高效的代码。

在 C++11 之前，标准里没有规定这样的东西，但 GCC、VC 等编译器发现这样做确实很有用，于是就实现出了自己“编译指令”；在 GCC 里是“`__attribute__`”，在 VC 里是“`__declspec`”。不过因为它们不是标准，所以名字显得有点“怪异”。

到了 C++11，标准委员会终于认识到了“编译指令”的好处，于是就把“民间”用法升级为“官方版本”，起了个正式的名字叫“**属性**”。你可以把它理解为给变量、函数、类等“贴”上一个编译阶段的“标签”，方便编译器识别处理。

“属性”没有新增关键字，而是用两对方括号的形式“`[[...]]`”，方括号的中间就是属性标签（看着是不是很像一张方方正正的便签条）。所以，它的用法很简单，比 GCC、VC 的都要简洁很多。

我举个简单的例子，你看一下就明白了：

```
1 [[noreturn]]           // 属性标签
2 int func(bool flag)    // 函数绝不会返回任何值
3 {
```

 复制代码

```
4     throw std::runtime_error("XXX");
5 }
```


不过，在 C++11 里只定义了两个属性：“noreturn”和“carries\_dependency”，它们基本上没什么大用处。

C++14 的情况略微好了点，增加了一个比较实用的属性“deprecated”，用来标记不推荐使用的变量、函数或者类，也就是被“废弃”。

比如说，你原来写了一个函数 old\_func()，后来觉得不够好，就另外重写了一个完全不同的新函数。但是，那个老函数已经发布出去被不少人用了，立即删除不太可能，该怎么办呢？

这个时候，你就可以让“属性”发挥威力了。你可以给函数加上一个“deprecated”的编译期标签，再加上一些说明文字：

```
1 [[deprecated("deadline:2020-12-31")]]      // C++14 or later
2 int old_func();
```

 复制代码

于是，任何用到这个函数的程序都会在编译时看到这个标签，报出一条警告：

```
1 warning: 'int old_func()' is deprecated: deadline:2020-12-31 [-Wdeprecated-dec]
```

 复制代码

当然，程序还是能够正常编译的，但这种强制的警告形式会“提醒”用户旧接口已经被废弃了，应该尽快迁移到新接口。显然，这种形式要比毫无约束力的文档或者注释要好得多。

目前的 C++17 和 C++20 又增加了五六个新属性，比如 fallthrough、likely，但我觉得，标准委员会的态度还是太“保守”了，在实际的开发中，这些真的是不够用。

好在“属性”也支持非标准扩展，允许以类似名字空间的方式使用编译器自己的一些“非官方”属性，比如，GCC 的属性都在“gnu::”里。下面我就列出几个比较有用的（全部属性可参考 [GCC 文档](#)）。

deprecated: 与 C++14 相同, 但可以用在 C++11 里。

unused: 用于变量、类型、函数等, 表示虽然暂时不用, 但最好保留着, 因为将来可能会用。

constructor: 函数会在 main() 函数之前执行, 效果有点像是全局对象的**构造函数**。

destructor: 函数会在 main() 函数结束之后执行, 有点像是全局对象的**析构函数**。

always\_inline: 要求编译器强制内联函数, 作用比 inline 关键字更强。


hot: 标记“热点”函数, 要求编译器更积极地优化。

这几个属性的含义还是挺好理解的吧, 我拿“unused”来举个例子。

在没有这个属性的时候, 如果有暂时用不到的变量, 我们只能用“(void) var;”的方式假装用一下, 来“骗”过编译器, 属于“不得已而为之”的做法。

那么现在, 我们就可以用“unused”属性来清楚地告诉编译器: 这个变量我暂时不用, 请不要过度紧张, 不要发出警告来烦我:

```
1  [[gnu::unused]]           // 声明下面的变量暂不使用, 不是错误
2  int nouse;
```

 复制代码

[GitHub 仓库](#)里的示例代码里还展示了其他属性的用法, 你可以在课下参考。

## 静态断言 (static\_assert)

“属性”像是给编译器的一个“提示”“告知”, 无法进行计算, 还算不上是编程, 而接下来要讲的“**静态断言**”, 就有点编译阶段写程序的味道了。

你也许用过 assert 吧, 它用来断言一个表达式必定为真。比如说, 数字必须是正数, 指针必须非空、函数必须返回 true:

```
1  assert(i > 0 && "i must be greater than zero");
2  assert(p != nullptr);
3  assert(!str.empty());
```

 复制代码

当程序（也就是 CPU）运行到 `assert` 语句时，就会计算表达式的值，如果是 `false`，就会输出错误消息，然后调用 `abort()` 终止程序的执行。

注意，`assert` 虽然是一个宏，但在预处理阶段不生效，而是在运行阶段才起作用，所以又叫“**动态断言**”。

有了“动态断言”，那么相应的也就有“静态断言”，名字也很像，叫“**`static_assert`**”，不过它是一个专门的关键字，而不是宏。因为它只在编译时生效，运行阶段看不见，所以是“静态”的。

“静态断言”有什么用呢？

类比一下 `assert`，你就可以理解了。它是编译阶段里检测各种条件的“断言”，编译器看到 `static_assert` 也会计算表达式的值，如果值是 `false`，就会报错，导致编译失败。

比如说，这节课刚开始时的斐波拉契数列计算函数，可以用静态断言来保证模板参数必须大于等于零：

 复制代码

```
1  template<int N>
2  struct fib
3  {
4      static_assert(N >= 0, "N >= 0");
5
6      static const int value =
7          fib<N - 1>::value + fib<N - 2>::value;
8  };
```

再比如说，要想保证我们的程序只在 64 位系统上运行，可以用静态断言在编译阶段检查 `long` 的大小，必须是 8 个字节（当然，你也可以换个思路用预处理编程来实现）。

 复制代码

```
1  static_assert(
2      sizeof(long) >= 8, "must run on x64");
3
4  static_assert(
```



这里你一定要注意，`static_assert` 运行在编译阶段，只能看到编译时的常数和类型，看不到运行时的变量、指针、内存数据等，是“静态”的，所以不要简单地把 `assert` 的习惯搬过来用。

比如，下面的代码想检查空指针，由于变量只能在运行阶段出现，而在编译阶段不存在，所以静态断言无法处理。

[复制代码](#)

```
1 char* p = nullptr;
2 static_assert(p == nullptr, "some error."); // 错误用法
```

说到这儿，你大概对 `static_assert` 的“编译计算”有点感性认识了吧。在用“静态断言”的时候，你就要在脑子里时刻“绷紧一根弦”，把自己代入编译器的角色，**像编译器那样去思考**，看看断言的表达式是不是能够在编译阶段算出结果。

不过这句话说起来容易做起来难，计算数字还好说，在泛型编程的时候，怎么检查模板类型呢？比如说，断言是整数而不是浮点数、断言是指针而不是引用、断言类型可拷贝可移动.....

这些检查条件表面上看好像是“不言自明”的，但要把它用 C++ 语言给精确地表述出来，可就没那么简单了。所以，想要更好地发挥静态断言的威力，还要配合标准库里的“`type_traits`”，它提供了对应这些概念的各种编译期“函数”。

[复制代码](#)

```
1 // 假设T是一个模板参数，即template<typename T>
2
3 static_assert(
4     is_integral<T>::value, "int");
5
6 static_assert(
7     is_pointer<T>::value, "ptr");
8
9 static_assert(
10    is_default_constructible<T>::value, "constructible");
11
```

你可能看到了，“static\_assert”里的表达式样子很奇怪，既有模板符号“<>”，又有作用域符号“::”，与运行阶段的普通表达式大相径庭，初次见到这样的代码一定会吓了一跳。

这也是没有办法的事情。因为 C++ 本来不是为编译阶段编程所设计的。受语言的限制，编译阶段编程就只能“魔改”那些传统的语法要素了：把类当成函数，把模板参数当成函数参数，把“::”当成 return 返回值。说起来，倒是和“函数式编程”很神似，只是它运行在编译阶段。

由于“type\_traits”已经初步涉及模板元编程的领域，不太好一下子解释清楚，所以，在这里我就不再深入介绍了，你可以课后再看看这方面的其他资料，或者是留言提问。

## 小结

好了，今天我和你聊了 C++ 程序在编译阶段能够做哪些事情。

编译阶段的“主角”是编译器，它依据 C++ 语法规则处理源码。在这个过程中，我们可以用一些手段来帮助编译器，让它听从我们的指挥，优化代码或者做静态检查，更好地为运行阶段服务。

但要当心，毕竟只有编译器才能真正了解 C++ 程序，所以我们还是要充分信任它，不要过分干预它的工作，更不要有意与它作对。

我们来小结一下今天的要点。

1. “属性”相当于编译阶段的“标签”，用来标记变量、函数或者类，让编译器发出或者不发出警告，还能够手工指定代码的优化方式。
2. 官方属性很少，常用的只有“deprecated”。我们也可以使用非官方的属性，需要加上名字空间限定。
3. static\_assert 是“静态断言”，在编译阶段计算常数和类型，如果断言失败就会导致编译错误。它也是迈向模板元编程的第一步。
4. 和运行阶段的“动态断言”一样，static\_assert 可以在编译阶段定义各种前置条件，充分利用 C++ 静态类型语言的优势，让编译器执行各种检查，避免把隐患带到运行阶段。

## 课下作业



最后是课下作业时间，给你留两个思考题：

1. 预处理阶段可以自定义宏，但编译阶段不能自定义属性标签，这是为什么呢？
2. 你觉得，怎么用“静态断言”，才能更好地改善代码质量？

欢迎你在留言区写下你的思考和答案，如果觉得今天的内容对你有所帮助，也欢迎把它分享给你的朋友。我们下节课见。

# 课外小贴士

1. 编译器自己的指令“\_\_attribute\_\_”“\_\_declspec”也有一点优势，不仅能够用在C++程序里，也能够用在C程序里。
2. 使用assert要包含头文件<cassert>。另外，如果在编译时定义了宏“NDEBUG”，就会令assert宏为空，完全禁用断言。
3. 不要依赖assert来检测或者预防错误，而是要把它作为一种“文档形式的代码”，显式地表明前提条件和后续结果。
4. C++17之后简化了“type\_traits”库里元函数的调用方式，不必再使用“xxx::value”“xxx::type”的形式获取返回值，直接用等价的别名形式“xxx\_v”“xxx\_t”就行了，例如“is\_void\_v<T>”等价于“is\_void<T>::value”。
5. C++20引入了concept，将来可能还会有contract，会让编译阶段编程更加简单易懂。
6. 虽然有“王婆卖瓜”的嫌疑，但如果你想要深入学习模板元编程，我还是推荐你看一下《C++11/14高级编程 Boost程序库探秘》这本书，里面介绍的内容很详细，也很完整。

# 6月-7月课表抢先看

## 充 ¥500 得 ¥580

赠「¥ 118 月球主题 AR 笔记本」



【点击】图片, 立即查看>>>

© 版权归极客邦科技所有, 未经许可不得传播售卖。页面已增加防盗追踪, 如有侵权极客邦将依法追究其法律责任。

上一篇 03 | 预处理阶段能做什么: 宏定义和条件编译

下一篇 05 | 面向对象编程: 怎样才能写出一个“好”的类?

### 精选留言 (22)

写留言



yelin

2020-05-14

斐布那契还可以这么玩, 期待老师后面对于模版类的课程, 我可能从来没都没学会过

作者回复: 模板元编程比较复杂, 属于屠龙之术, 这次我先不讲, 如果感兴趣的同学多可以以后单独开一个课程。

2

8



Luca

2020-05-14

1. 因为属性标签都在编译器里内置, 自定义的属性标签编译器无法识别。

2. 静态断言可以作为编译期的一种约定，配合错误提示能够更快发现编译期的错误。

作者回复: very nice。



7



**lckfa李钊**

2020-05-14

看到老师的斐波那契数列实现，我还是挺惊讶的，代码虽都看得懂，但是从没想到这么写，我有两个问题想请教下：1.按本节的主题，编译阶段能做什么，所以说后面的那几个斐波那契数列在编译器就有结果了吗？如果是这样的话，肯定是需要cpu压栈计算的，这和真实的运行期有哪些不同呢？2.模板编程在哪些场景下使用比较好？模板编程和编译阶段似乎关联更大些

展开 ∨

作者回复:

1.是的，这些代码都是模板类，自然会由编译器去解析处理，最后出来的也是编译期数值，也就是静态常量，省去了运行期的技术成本，运行期直接用就行。

2.模板元编程和预处理编程有点像，由编译器来改变源码的形态，但它的规则更复杂，难以理解，你首先要了解泛型编程，之后才能尝试模板元编程。

对于80%的C++程序员来说，我不建议尝试模板元编程，可以参考第1讲。



3



**EncodedStar**

2020-05-19

老师可以在每讲开始讲讲上一讲提到的问题吗？很多疑惑~

用“静态断言”，是不是在代码严格要求是32位系统或者64位系统的时候也比较有用呢？32位系统和64位系统本身有的类型所占字节数不同。

展开 ∨

作者回复:

1.课程都是预先录好的，所以不能及时回答，有问题写在留言里，我可以回复，还是希望自己思考得到答案。

2.静态断言的用处很多，判断32/64只是个最简单的例子，只要能够在编译阶段计算出的结果就可以断言，不过这就需要对编译阶段有比较多的认识了。

不用着急，慢慢学C++，了解了泛型后再看静态断言可能会好懂一些。



**EncodedStar**

2020-05-18

预处理可以自定义是直接将定义好的内容写到源码里，而标签不能自定义是因为编译器需要识别标签名

作者回复: good



**Carlos**

2020-05-14

不得不说这节课让我回忆起了自己刚学会 vim macros 的感觉: 原来是我的想象力限制了 vim... 现在我想说: 原来是我的想象力限制了 c++... 🤔

今天两个问题我都不不是很懂, 希望老师指正.

...

展开 ∨

作者回复:

1.回答沾点边。实际上是因为属性标签必须要由编译器解释，而自定义标签编译器是不认识的，所以只能等编译器开发者去加，而不能是自己加。

2.说的比较好。

静态断言是一种对编译环境的“前提”“假设”，要求在编译阶段必须如何如何，可以结合第1讲的生命周期，考虑一下应该如何发挥它的作用。



**逸清**

2020-05-14

老师，自己C++基础知识还算了解，但代码写的太少，拿到一个需求无从下手，老师有没有比较好的方法或者适合练手的项目推荐？

展开 ∨

作者回复: 建议先学习一下标准库，了解里面的那些工具，现在开发很少有白手起家的了，用好工具，知道它们能解决哪些问题，写应用也就比较容易了。

比如string/regex处理字符串、map/set集合、线程库等等，跟着课程逐步学吧。



jxon-H

2020-05-20

第三次学习这节课的内容，感觉自己总算明白了罗老师的苦心。

与一般的C++课不同，罗老师完全不讲语法要素这些百度一大把，而是从工作的原理和本质去剖析C++。

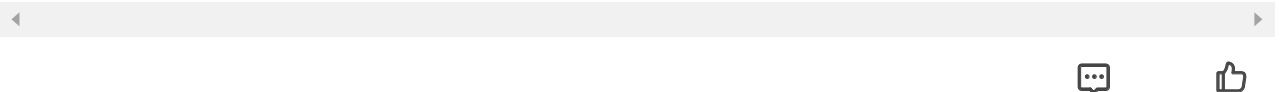
我记得开课的第一讲，罗老师就这么说过，当时没啥体会，现在越发觉得这样的编排确实很高级。...

展开 ∨

作者回复: 有点过誉了，受之有愧。

因为C++比较复杂，所以我划出了四个生命周期，方便特性的归类和理解，不然混在一起很容易把思路弄乱。

C++需要在实践中学，要花的时间和精力还是挺多的，不过乐趣也自在其中。



~灯火阑珊

2020-05-17

1.之前碰过一道面试题问：C++ assert的断言是怎么实现的？如何编写跨平台的断言函数？.我当时答的调用abort，感觉不好。这里不是可以从静态断言和动态断言两个方面答啊。老师对于这到面试题可不可以给点思路？

2.文中"static\_assert 可以在编译阶段定义各种前置条件，充分利用 C++ 静态类型语言...

展开 ∨

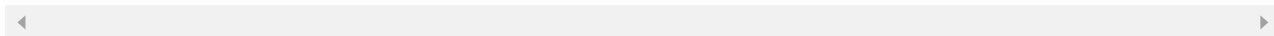
作者回复:

1.这面试题太细节了，我也没有深究过，觉得这个题没什么意义。你的思路我觉得靠谱，可以展开来说。

2.在编译期断言各种条件，比如必须是64位平台，类型必须是指针，类型必须有某个成员函数，类型必须可以拷贝等等，需要有编译期的思维方式。

3.是相当于动态语言而言的，比如Python、php，变量类型是动态的。





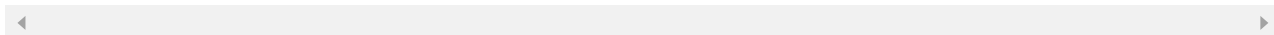
**silverhawk**

2020-05-17

属性这个，比起Java，python差好多，有了属性可以编译器静态的检查很多OO编程，比如override之类的对不对啊

展开 ∨

作者回复: 目前的C++11这块的确很弱，没办法，标准委员会效率就是低，不像公司那样无阻力大干快上。



**幻境之桥**

2020-05-17

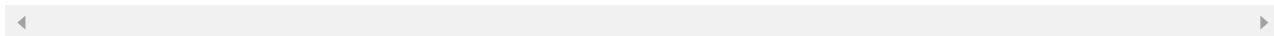
1 预处理阶段的宏是我们自己来处理的，标签是编译器来处理，除非开发拓展编译器才需要或可以定义标签

2 static\_assert 可以在编译时检查是否满足编译环境要求，不满足直接编译失败，static\_const 也类似吧！

展开 ∨

作者回复: 说的很好。

不过后面的static\_const不知道是什么，没这个关键字。



2



**有学识的兔子**

2020-05-16

1. 预处理不受编译器控制，由预处理器负责，给予宏的自由度比较大；而属性标签是为了简化编译器工作，而非为了扩展；

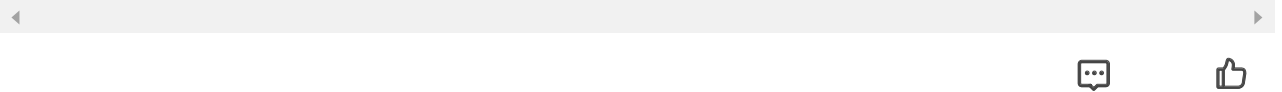
2. 例如在编译阶段检查x64还得x86，跟常量运算有关的逻辑判断。

展开 ∨

作者回复:

1.预处理器不需要理解宏，只是文本替换，而编译器必须要理解属性才能处理，自定义属性标签相当于是“外来语”，识别不了编译器就无法工作。

2.这个只是最基本的用法，随着对C++理解的深入，还可以对类型做各种静态检查。



**李学文 Alvin**

2020-05-15

希望老师推荐一款Ubuntu16.4下C++编辑器或IDE，谢谢 人

作者回复: 我常用的就是vim，你可能不太习惯，其他的没用过，抱歉。

可以参考其他同学，用vs code，然后用插件远程登录Linux。



**tt**

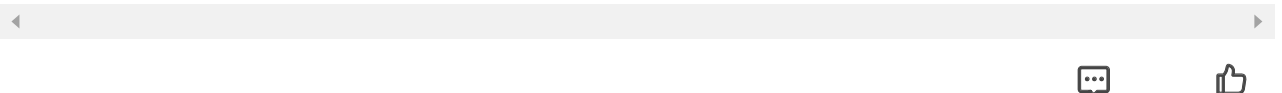
2020-05-15

受语言的限制，编译阶段编程就只能“魔改”那些传统的语法要素了：把类当成函数，把模板参数当成函数参数，把“::”当成return返回值。

这个说法真形象，那些乱七八糟的语法一下就不面目可憎了。

展开 v

作者回复: 嗯，这也是我反复思考才得出的经验。



**J淡忘**

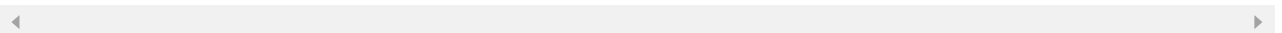
2020-05-15

在vs中使用 标记 deprecated 的方法 好像会报错  
c4996

加上 宏可以解除报错 但也没有警告  
不知道老师有没有办法

作者回复: 没用过vs，不好帮你。

不过我觉得上网搜一下，应该是个常见的问题，有解决方案。





**Tedeer**

2020-05-14

老师，因为在做Android时，会做一些java层的反编译；很少做so库的反编译，我很好奇so反编译生成的代码还会有这些属性标签和断言吗？

展开 ∨

作者回复: Android和Java不太熟，不是很了解。但我觉得，属性和断言都是源码级别的，如果反编译这些信息应该是看不见的。



**忆水寒**

2020-05-14

学习了不少知识，期待老师能多讲讲C++ 一些在项目实战中 比较好用的方法。

作者回复: 后面还有很多我在实际中的经验总结，可以慢慢看，有什么地方没讲到的也可以提，知无不言言无不尽。



**alioo**

2020-05-14

老师好，我看前面的回答是说template是在编译阶段完成的，但是我使用g++ compile.cpp -E发现并没有像上一节的宏，宏直接就计算出值了，然而template却没有计算

作者回复: 注意，-E处理的是预处理阶段的宏，而template是编译阶段，这是两个完全不同的阶段。

编译阶段出来的结果直接就是二进制码了，是看不到源码的。



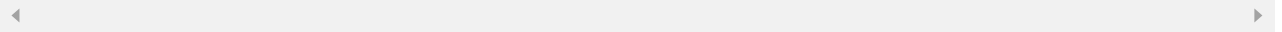
**范闲**

2020-05-14

1. 标签内置在编译器内部，无法进行自定义。
2. 断言在编译期间配合错误检查，能提前发现代码漏洞。

展开 ∨

作者回复: great。



**Seven**

2020-05-14

属性名和Java中的相似

展开 ∨

作者回复: 所谓英雄所见略同。

